Network Applications: High-performance Server Design & Operational Analysis

Y. Richard Yang

http://zoo.cs.yale.edu/classes/cs433/

10/5/2017
Assignment three (Part 2) to be posted later today.

Date for exam 1?
- Oct. 19, 24, 26?
Recap: Thread-Based Network Servers

- Why: blocking operations; threads (execution sequences) so that only one thread is blocked

- How:
  - Per-request thread
    - problem: large # of threads and their creations/deletions may let overhead grow out of control
  - Thread pool
    - Design 1: Service threads compete on the welcome socket
    - Design 2: Service threads and the main thread coordinate on the shared queue
      - polling (busy wait)
      - suspension: wait/notify
Use Java ThreadPoolExecutor

server = new ServerSocket(port);
System.out.println("Time server listens at port: " + port);

// Create Java Executor Pool
TimeServerHandlerExecutePool myExecutor = new TimeServerHandlerExecutePool(50, 10000);

Socket socket = null;
while (true) {
    socket = server.accept();
    myExecutor.execute(new TimeServerHandler(socket));
} // end of while
public class TimeServerHandlerExecutePool {

    private ExecutorService executor;

    public TimeServerHandlerExecutePool(int maxPoolSize, int queueSize) {
        executor = new ThreadPoolExecutor(
            Runtime.getRuntime().availableProcessors(),
            maxPoolSize,
            120L, TimeUnit.SECONDS,
            new ArrayBlockingQueue<java.lang.Runnable>(queueSize)
        );
    }

    public void execute(java.lang.Runnable task) {
        executor.execute(task);
    }
}

For Java ThreadPoolExecutor scheduling algorithm, see:
https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ThreadPoolExecutor.html
Recap: Program Correctness Analysis

- Safety
  - consistency
  - app requirement, e.g., Q.remove() is not on an empty queue

- Liveness (progress)
  - main thread can always add to Q
  - every connection in Q will be processed

- Fairness
  - For example, in some settings, a designer may want the threads to share load equally
Summary: Using Threads

- **Advantages**
  - Intuitive (sequential) programming model
  - Shared address space simplifies coordination

- **Disadvantages**
  - Overhead: thread stacks, synchronization
  - Thread pool parameter (how many threads) difficult to tune
Outline

- Admin and recap
- High performance servers
  - Threaded design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design
Basic Idea: Multiplexed, Reactive I/O

- **Approach:** peek system state, issue function calls only for those who are ready
  - also called reactive (Reactor design), or multiplexed non-blocking

![Diagram of server state and completed connections]

- **Server**
  - 128.36.232.5
  - 128.36.230.2

- **TCP socket space**

- **Completed connection**
  - sendbuf full or has space
  - recvbuf empty or has data

- **Completed connection queue**
  - C1; C2
Multiplexed, Reactive Server I/O Support

- Modern operating systems, such as Windows, Mac and Linux, provide facilities for fast, scalable IO based on the use of **notifications** of ready IO operations:
  - Linux: select, epoll (2.6)
  - Mac/FreeBSD: kqueue
Basic Idea: Asynchronous Initiation and Callback

- Issue of only peek:
  - Cannot handle initiation calls (e.g., read file, initiate a connection by a network client)

- Idea: asynchronous initiation (e.g., aio_read) and program specified completion handler (callback)
  - Also referred to as proactive (Proactor) nonblocking

- We focus more on multiplexed, reactive design
Outline

- Admin and recap
- High performance servers
  - Thread design
    - Per-request thread
    - Thread pool
      - Busy wait
      - Wait/notify
  - Asynchronous design
    - Overview
    - Multiplexed (selected), reactive programming
Program registers events (e.g., acceptable, readable, writable) to be monitored and a handler to call when an event is ready

An infinite dispatcher loop:
- Dispatcher asks OS to check if any ready event
- Dispatcher calls (multiplexes) the registered handler of each ready event/source
  - Handler should be non-blocking, to avoid blocking the event loop
Multiplexed, Non-Blocking Network Server

// clients register interests/handlers on events/sources
while (true) {
  - ready events = select()
    /* or selectNow(),
       or select(int timeout) to check ready events from the registered interests */

  - foreach ready event {
      switch event type:
      accept: call accept handler
      readable: call read handler
      writable: call write handler
    }

  - handle other events
}
Main Abstractions

- Main abstractions of multiplexed IO:
  - **Channels**: represent connections to entities capable of performing I/O operations;
  - **Selectors and selection keys**: selection facilities;
  - **Buffers**: containers for data.

- More details see [https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html](https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html)
Multiplexed (Selectable), Non-Blocking Channels

<table>
<thead>
<tr>
<th>Channel Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SelectableChannel</td>
<td>A channel that can be multiplexed</td>
</tr>
<tr>
<td>DatagramChannel</td>
<td>A channel to a datagram-oriented socket</td>
</tr>
<tr>
<td>Pipe.SinkChannel</td>
<td>The write end of a pipe</td>
</tr>
<tr>
<td>Pipe.SourceChannel</td>
<td>The read end of a pipe</td>
</tr>
<tr>
<td>ServerSocketChannel</td>
<td>A channel to a stream-oriented listening socket</td>
</tr>
<tr>
<td>SocketChannel</td>
<td>A channel for a stream-oriented connecting socket</td>
</tr>
</tbody>
</table>

- **Use** `configureBlocking(false)` **to make a channel non-blocking**
- **Note:** Java `SelectableChannel` **does not include file I/O**
Selector

- The class `Selector` is the base of the multiplexer/dispatcher
- Constructor of Selector is protected; create by invoking the `open` method to get a selector (why?)
Selector and Registration

- A selectable channel registers events to be monitored with a selector with the register method

- The registration returns an object called a SelectionKey:

```java
SelectionKey key = channel.register(selector, ops);
```
A SelectionKey object stores:

- **interest set**: events to check: `key.interestOps(ops)`
- **ready set**: after calling `select`, it contains the events that are ready, e.g. `key.isReadable()`
- **an attachment** that you can store anything you want: `key.attach(myObj)`
Checking Events

- A program calls `select` (or `selectNow()`, or `select(int timeout)`) to check for ready events from the registered SelectableChannels
  - Ready events are called the selected key set
    ```java
    selector.select();
    Set readyKeys = selector.selectedKeys();
    ```

- The program iterates over the selected key set to process all ready events
Dispatcher using Select

```javascript
while (true) {
    - selector.select()
    - Set readyKeys = selector.selectedKeys();

    - foreach key in readyKeys {
        switch event type of key:
            accept: call accept handler
            readable: call read handler
            writable: call write handler
    }
}
```
I/O in Java: ByteBuffer

- Java SelectableChannels typically use ByteBuffer for read and write
  - channel.read(byteBuffer);
  - channel.write(byteBuffer);

- ByteBuffer is a powerful class that can be used for both read and write
- It is derived from the class Buffer
## Java ByteBuffer Hierarchy

<table>
<thead>
<tr>
<th>Buffers</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Position, limit, and capacity; clear, flip, rewind, and mark/reset</td>
</tr>
<tr>
<td>ByteBuffer</td>
<td>Get/put, compact, views; allocate, wrap</td>
</tr>
<tr>
<td>MappedByteBuffer</td>
<td>A byte buffer mapped to a file</td>
</tr>
<tr>
<td>CharBuffer</td>
<td>Get/put, compact; allocate, wrap</td>
</tr>
<tr>
<td>DoubleBuffer</td>
<td>' '</td>
</tr>
<tr>
<td>FloatBuffer</td>
<td>' '</td>
</tr>
<tr>
<td>IntBuffer</td>
<td>' '</td>
</tr>
<tr>
<td>LongBuffer</td>
<td>' '</td>
</tr>
<tr>
<td>ShortBuffer</td>
<td>' '</td>
</tr>
</tbody>
</table>
Buffer (relative index)

- Each Buffer has three numbers: position, limit, and capacity
  - Invariant: $0 \leq position \leq limit \leq capacity$

- Buffer.clear(): position = 0; limit = capacity
channel.read(Buffer)

- Put data into Buffer, starting at position, not to reach limit
channel.write(Buffer)

- Move data from Buffer to channel, starting at position, not to reach limit
Buffer.flip(): limit=position; position=0

Why flip: used to switch from preparing data to output, e.g.,

- buf.put(header); // add header data to buf
- in.read(buf); // read in data and add to buf
- buf.flip(); // prepare for write
- out.write(buf);

Typical pattern: read, flip, write
Buffer.compact()

- Move \([\text{position}, \text{limit})\) to 0
- Set position to \(\text{limit} - \text{position}\), limit to \(\text{capacity}\)

// typical design pattern
buf.clear(); // Prepare buffer for use
for (;;) {
  if (in.read(buf) < 0 && !buf.hasRemaining())
    break; // No more bytes to transfer
  buf.flip();
  out.write(buf);
  buf.compact(); // In case of partial write
}
Example

- See NioEchoServer/v1-2/EchoServer.java
Problems of Echo Server v1

- Empty write: Callback to `handleWrite()` is unnecessary when nothing to write
  - Imagine empty write with 10,000 sockets
  - Solution: initially read only, later allow write

- `handleRead()` still reads after the client closes
  - Solution: after reading end of stream (read returns -1), deregister read interest for the channel
Still many remaining issues such as idle instead of close
Why no need to introduce FSM for a thread version?

FSM for each socket channel in v2:

- Idle
  - Read
  - Write
  - Read close

Accept Client Connection
- Read Request
- Find File
  - Send Response Header
    - Read File
    - Send Data

Finite-State Machine and Thread
A More Typical Finite State Machine

Read from client channel

- !RequestReady
- !ResponseReady
- InitInterest = READ

Request complete (find terminator or client request close)

Generating response

- RequestReady
- !ResponseReady
- Interest = -

Write response

- Closed
- Interest = -

ResponseSent

- RequestReady
- ResponseReady
- ResponseReady
- Interest = Write
There can be multiple types of FSMs, to handle protocols correctly.

- Staged: first read request and then write response
- Mixed: read and write mixed

Choice depends on protocol and tolerance of complexity, e.g.,

- HTTP/1.0 channel may use staged
- HTTP/1.1/2/Chat channel may use mixed
Toward More General Server Framework

- Our example EchoServer is for a specific protocol.

- A general non-blocking, reactive programming framework tries to introduce structure to allow substantial program reuse.
  - Non-blocking programming framework is among the more complex software systems.
  - We will see one simple example, using EchoServer as a basis.
A More Extensible Dispatcher Design

- Fixed accept/read/write functions are not general design
  - A solution: Using attachment of each channel
    - Attaching a `ByteBuffer` to each channel is a narrow design for simple echo servers
    - A more general design can use the attachment to store a callback that indicates not only data (state) but also the handler (function)
A More Extensible Dispatcher Design

- Attachment stores generic event handler
  - Define interfaces
    - IAcceptHandler and
    - IReadWriteHandler
  - Retrieve handlers at run time

```java
if (key.isAcceptable()) { // a new connection is ready
    IAcceptHandler aH = (IAcceptHandler) key.attachment();
    aH.handleAccept(key);
}

if (key.isReadable() || key.isWritable()) {
    IReadWriteHandler rwH = IReadWriteHandler)key.attachment();
    if (key.isReadable()) rwH.handleRead(key);
    if (key.isWritable()) rwH.handleWrite(key);
}
```
Handler Design: Acceptor

What should an accept handler object know?

- ServerSocketChannel (so that it can call accept)
  - Can be derived from SelectionKey in the call back

- Selector (so that it can register new connections)
  - Can be derived from SelectionKey in the call back

- What ReadWrite object to create (different protocols may use different ones)?
  - Pass a Factory object: SocketReadWriteHandlerFactory
Handler Design: ReadWriteHandler

- What should a ReadWrite handler object know?
  - SocketChannel (so that it can read/write data)
    - Can be derived from SelectionKey in the call back
  - Selector (so that it can change state)
    - Can be derived from SelectionKey in the call back
Class Diagram of SimpleNAIO

- **Dispatcher**
  - ...

- **IChannelHandler**
  - handleException();

- **IAcceptHandler**
  - handleAccept();

- **Accept**
  - implements

- **ISocketReadWriteHandlerFactory**
  - createHandler();

- **EchoReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **EchoReadWriteHandlerFactory**
  - createHandler();

- **IReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();
Class Diagram of SimpleNAIO

- **Dispatcher**
  - ...

- **IChannelHandler**
  - handleException();

- **IAcceptHandler**
  - handleAccept();

- **ISocketReadWriteHandlerFactory**
  - createHandler();

- **Accept**
  - implements

- **NewReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **NewReadWriteHandlerFactory**
  - createHandler();

- **IReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **EchoReadWriteHandler**
  - handleRead();
  - handleWrite();
  - getInitOps();

- **EchoReadWriteHandlerFactory**
  - createHandler();
SimpleNAIO

- See NioEchoServer/v3/*\.java
Discussion on SimpleNAIO

- In our current implementation (Server.java)

1. Create dispatcher
2. Create server socket channel and listener
3. Register server socket channel to dispatcher
4. Start dispatcher thread

Can we switch 3 and 4?
Extending SimpleNAIO

- A production network server often closes a connection if it does not receive a complete request in TIMEOUT.

- One way to implement time out is that
  - the read handler registers a timeout event with a timeout watcher thread with a call back
  - the watcher thread invokes the call back upon TIMEOUT
  - the callback closes the connection

Any problem?
Extending Dispatcher Interface

- Interacting from another thread to the dispatcher thread can be tricky
- Typical solution: async command queue

```java
while (true) {
    - process async. command queue
    - ready events = select (or selectNow(), or select(int timeout)) to check for ready events from the registered interest events of SelectableChannels

    - foreach ready event
      call handler
}
```
Question

How may you implement the async command queue to the selector thread?

```java
public void invokeLater(Runnable run) {
    synchronized (pendingInvocations) {
        pendingInvocations.add(run);
    }
    selector.wakeup();
}
```
What if another thread wants to wait until a command is finished by the dispatcher thread?
public void invokeAndWait(final Runnable task)
    throws InterruptedException {
    if (Thread.currentThread() == selectorThread) {
        // We are in the selector's thread. No need to schedule
        // execution
        task.run();
    } else {
        // Used to deliver the notification that the task is executed
        final Object latch = new Object();
        synchronized (latch) {
            // Uses the invokeLater method with a newly created task
            this.invokeLater(new Runnable() {
                public void run() {
                    task.run();
                    // Notifies
                    synchronized (latch) { latch.notify(); }
                }
            });
            // Wait for the task to complete.
            latch.wait();
        }
        // Ok, we are done, the task was executed. Proceed.
    }
}
Backup Slides
Events obscure control flow
- For programmers and tools

### Threads
```c
void thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    read_request(&s);
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

void pin_cache(struct session *s) {
    pin(&s);
    if (!in_cache(&s))
        read_file(&s);
    unpin(&s);
}
```

### Events
```c
void AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
}

void RequestHandler(struct session *s) {
    ...; CacheHandler.enqueue(s);
}

void CacheHandler(struct session *s) {
    pin(s);
    if (!in_cache(s))
        ReadFileHandler.enqueue(s);
    else                    ResponseHandler.enqueue(s);
}

void ExitHandler(struct session *s) {
    ...; unpin(&s); free_session(s); }
```
State Management

- Events require manual state management
- Hard to know when to free
  - Use GC or risk bugs

### Threads

```c
thread_main(int sock) {
    struct session s;
    accept_conn(sock, &s);
    if( !read_request(&s) )
        return;
    pin_cache(&s);
    write_response(&s);
    unpin(&s);
}

pin_cache(struct session *s) {
    pin(&s);
    if( !in_cache(s) ) ReadFileHandler.enqueue(s);
    else ResponseHandler.enqueue(s);
}
```

### Events

```c
CacheHandler(struct session *s) {
    pin(s);
    if( !in_cache(s) ) ReadFileHandler.enqueue(s);
    else ResponseHandler.enqueue(s);
}

RequestHandler(struct session *s) {
    ...; if( error ) return; CacheHandler.enqueue(s);
}

ExitHandler(struct session *s) {
    ...; unpin(&s); free_session(s);
}

AcceptHandler(event e) {
    struct session *s = new_session(e);
    RequestHandler.enqueue(s);
}
```

[von Behren]